# Presenting overrepresented words

Daniel J. H. Nebdal

December 1, 2008

# Contents

# 1 Summary

The genome of any living organism is the information stored in its DNA: A series of base pairs, usually represented as a string where each letter is one of A,T,G,C and represents one base. Advances in technology has made it possible and reasonably convenient to extract the full genome of an organism, and thus we now have this information for a number of species.

There are several ways of making this raw series of bases useful. One way is to consider what it codes for and what this means chemically and biologically. Another possibility is to do statistics on it *as text*, and then evaluating the results from a biological point of view.

Our work is in the latter camp. Specifically, one possible statistical analysis is to consider words: Short strings of bases (in our case, usually 8). Since DNA isn't a randomly generated text, it comes as no surprise that some words are more common than others. Even with that in mind, some species[1] have a few words abundant to a degree not commonly seen — finding these and their overrepresented words can be biologically interesting.

One of the explanations for these is that they are DNA Uptake Sequences, a kind of marker that lets the bacteria recognise a string of DNA as coming from itself or other closely related bacteria.

In addition to these special cases, we hoped that it would be possible to detect words with a much lower degree of overrepresentation, perhaps by correlating their positions on the genome with that of other known units, such as genes.

Counting all n-base subwords of a large set of known genomes and doing statistics on them has been done; my task has been to find a way to present these results and make them useful to an audience that might not have the time, technical skills or inclination to parse the raw numbers, and to add an extra level of analysis and crossreferencing.

This has resulted in a software package consisting of a web application for browsing and searching the data, and a set of tools that can be used by a person with some programming skill to look for other correlations and statistics.

[1] All of them bacteria, e.g. Haemophilus Influenzae

# 2 Relevant biology

## 2.1 Sequencing DNA

### 2.1.1 Reading directions

A DNA molecule is constructed from two strands, each strand carrying a series of bases. Each base has a complementary base that it will always be paired with, so an A always matches a T, and each G matches a C. This means that a transcription of one strand is sufficient, since the other strand can be reconstructed from that data.

This solves the problem of which strand to use: Either one will do. Defining a common reading direction is also fairly easy. A strand of DNA has two different ends (the 5' end and the 3' end), and all natural processes work in the 5' → 3' direction. Not surprisingly, this has also become the standard reading direction when transcribing DNA.

In a DNA molecule, the two strands lie in opposite directions, so each end of the complete molecule has a 5' end and a 3' end. When searching for a word in DNA, we generally want to search both strands. To do this, it is necessary to reverse the word (since the opposing strand has the opposite reading direction), and replace each base with its complement.


Figure 2.1: DNA reading directions

Using figure 2.1, it's obvious how each occurance of `AAGTG` implies `CACTT` on the other strand. A search for one would also need to include the other, to get matches on both strands.

# 3 Overrepresentation

## 3.1 Why

### 3.1.1 Genetic Uptake Sequences

Normally, a single strand of DNA floating around in a bacteria (or indeed any cell) will be dismantled quickly. These are either the result of some accident internal to the cell (such as a plasmid or chromosome breaking apart, or a problem during replication), or they come from outside the cell. Single strands might be harmful, by interfering during replication of transcription, or even by coding for a virus that could infect the cell — a well-function protection system is a definite advantage.

However, some bacteria complement this system by having a method for recognising DNA from other closely related bacteria, by the means of the DNA uptake sequences. These bacteria have a short and common pattern spread through their genome, and when they run into a DNA strand with it, it has a chance of being integrated into their genome.

Note that some families of bacteria possess the ability to take up any foreign DNA, and that this is orthogonal to having DNA uptake sequences. In the case where a species have both an uptake sequence and the ability to take up foreign material, material with a compatible uptake sequence will stand a greater change to be integrated. [AST$^+$84]

As an example of how common these uptake sequences are in the genomes that have them, we can look at Hamophilus Influenzae [1]. The three most overrepresented words of length 8 are `AGTGCGGT`, `AAGTGCGG`, and `AAAGTGCG`, with between 1669 and 1341 copies. If we consider this to be 1341 copies of a 10-base pattern, that means it covers about 0.73% of the 1.84 million bases.

### 3.1.2 Other causes of overrepresentation

While the uptake sequences are by far the most over-represented words we are aware of, they don't have a monopoly. In all bacteria (indeed, all genomes), there are other repeated features that might show up. The most likely one is the promoter regions that are often found in front of genes. These regions are recognised by the transcriptase protein that transcribes the genes to mRNA, and thus exist in multiple and fairly well preserved copies.

As an example, the Pribnow box is a motif commonly found ten bases before a gene.[Pri75][HR87] The consensus sequence is `TATAAT` , and searching for overrepresented words containing `*TATAAT*` yields a large list of results (see figure 3.1). Not all

[1] Specifically, these numbers are from Haemophilus influenzae PittEE, NC_009566

of these are *over*-represented; in some genomes they are significantly underrepresented, for reasons I'm not going to speculate on.

| Sequence ID | Word | Counted | Expected | E | Name |
|---|---|---|---|---|---|
| NC_007776 | ATATTATA | 43 | 2.658994 | 1.25e-17 | Synechococcus sp. JA-2-3B'a(2-13), complete genome. |
| NC_007776 | GTATAATA | 42 | 3.988311 | 3.83e-11 | Synechococcus sp. JA-2-3B'a(2-13), complete genome. |
| NC_007776 | ATTATACA | 47 | 5.262222 | 2.88e-10 | Synechococcus sp. JA-2-3B'a(2-13), complete genome. |
| NC_007606 | ATTATACA | 316 | 71.3425 | 2.453e-08 | Shigella dysenteriae Sd197, complete genome. |
| NC_009840 | GTATAATA | 10 | 95.42601 | 1.23e-07 | NO INFORMATION |
| NC_007775 | ATATTATA | 26 | 2.287103 | 1.43e-07 | Synechococcus sp. JA-3-3Ab, complete genome. |
| NC_007606 | CATTATAC | 355 | 85.83451 | 1.46e-07 | Shigella dysenteriae Sd197, complete genome. |
| NC_007332 | ATTATACC | 136 | 42.09439 | 2.032e-07 | Mycoplasma hyopneumoniae 7448, complete genome. |
| NC_005072 | ATTATAAC | 21 | 138.0554 | 3.313e-07 | Prochlorococcus marinus subsp. pastoris str. CCMP1986, complete |
| NC_008817 | GTATAATA | 15 | 107.3274 | 4.234e-07 | Prochlorococcus marinus str. MIT 9515, complete genome. |
| NC_009523 | ATTATAGC | 225 | 61.30306 | 2.683e-06 | Roseiflexus sp. RS-1, complete genome. |
| NC_009091 | GTATAATA | 12 | 86.56972 | 3.25e-06 | Prochlorococcus marinus str. MIT 9301, complete genome. |
| NC_008816 | ATTATAGC | 10 | 78.72819 | 3.476e-06 | Prochlorococcus marinus str. AS9601, complete genome. |
| NC_007577 | ATTATAAT | 39 | 199.0079 | 6.068e-06 | Prochlorococcus marinus str. MIT 9312, complete genome. |
| NC_009091 | GTATTATA | 13 | 87.82899 | 6.179e-06 | Prochlorococcus marinus str. MIT 9301, complete genome. |
| NC_005072 | GTATAATA | 17 | 100.3779 | 6.991e-06 | Prochlorococcus marinus subsp. pastoris str. CCMP1986, complete |
| NC_005072 | GTATTATA | 18 | 103.6526 | 8.445e-06 | Prochlorococcus marinus subsp. pastoris str. CCMP1986, complete |
| NC_009767 | ATTATACC | 205 | 60.18822 | 9.254e-06 | NO INFORMATION |
| NC_007295 | ATTATACC | 125 | 42.13343 | 1e-05 | Mycoplasma hyopneumoniae J, complete genome. |
| NC_009091 | ATTATACC | 9 | 69.6828 | 1.1e-05 | Prochlorococcus marinus str. MIT 9301, complete genome. |
| NC_008816 | ATTATAAT | 36 | 184.1267 | 1.4e-05 | Prochlorococcus marinus str. AS9601, complete genome. |
| NC_008817 | ATTATACC | 13 | 80.83389 | 1.4e-05 | Prochlorococcus marinus str. MIT 9515, complete genome. |
| NC_006360 | ATTATACC | 123 | 42.26118 | 1.7e-05 | Mycoplasma hyopneumoniae 232, complete genome. |
| NC_009009 | GTATAATA | 150 | 55.49966 | 1.9e-05 | Streptococcus sanguinis SK36, complete genome. |

Figure 3.1: Over- and underrepresented words containing the Pribnow box motif. The "NO INFORMATION" was caused by some tables containing newer data than others at the time.

### 3.1.3 How to determine

This section leans to a large degree on work done by Einar Rødland, both [Rod06], an as of yet unpublished article[2], and e-mail correspondence. Anyone interested in the mathematics should read [Rod06], this overview skips most of the calculations and several of the considerations.

#### How to model

In order to determine if a word is more (or indeed less) common than we expect it to be, we first need to agree on how common we *expect* it to be. This seems obvious, but the details are complicated and interesting.

The most trivial model is to assume a completely even and random base distribution. The expected frequency of an N-letter word would then be $(\frac{1}{4})^N$. This can be made more precise by using the observed AT/GC ratio from the genome, but still breaks down quickly — DNA is far from random, especially not the gene-heavy bacterial genomes. [Fit83]

A more sofisticated model is to model the genome as a Markov chain. A Markov chain of order N is a set of probabilities, where the probability of a given base in a

[2] A method for statistical assessment of over-represented words in DNA sequences

given position is a function of the N previous bases. As an example: If we wanted to model a genome with long repeats of `ATAT...` with a chain of order 2, we could set the probability of `A` after `AT`, and `T` after `TA` to be high.

One of the problems with a Markov model is that estimating these probabilities by counting transitions in the genome gives just that: An estimate. In addition, such a model can produce long base sequences that would never occur in an actual genome. A way of getting a more precise simulation is to use a variant of Markov chains where the total count of each word of length k is identical to that of the original genome — this is a *conditional Markov chain*, and the sequences they can produce can be seen as shuffled variants of the original genome.[Fit83]

For the sake of simplicity, it's easiest to consider all genomes as circular, so the last base is followed by the first; in the case of a linear genome we can add an unique delimiter sybol between them.

One way of representing the possible shuffled variants is by looking to graph theory. Each k-letter word can be represented as a directed edge between two words of length k-1 — e.g `ABCD` as `ABC`→`BCD`. The original genome is then a path through this graph, using each edge once and ending at the same node it started as. A path like that is an Euler cycle, and all complete shufflings have this form.

If we construct shufflings by randomly pairing outgoing and incoming edges, many of them won't be complete Euler cycles, but instead sets of smaller unconnected cycles. Including these as well as the full cycles simplifies the calculations at no significant cost in precision, so the following calculations do that.

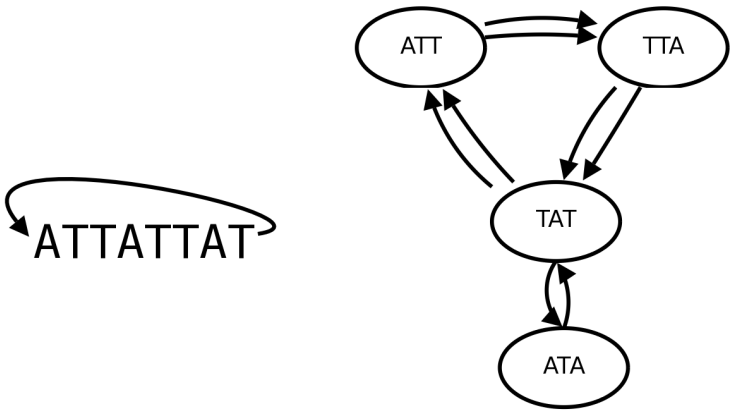


Figure 3.2: A very short genome and the corresponding directed graph

If we want to consider the word $w$ in the graph $G$, we first construct a path $v$ in $G$ corresponding to it. We then construct a new graph $G_v$ by replacing the path $v$ with a single edge from the first to the last node in it — se figure 3.3. Each possible Euler cycle in $G_v$ will then correspond to a cycle in $G$ that contains $w$.

There are multiple ways of constructing $v$ in a given Euler cycle $\gamma$, and multiple ways

to pick $\gamma$ in the graph $G$. If we count the number of possible pairs of $(\gamma, v)$ where $\gamma$ is an Euler cycle in $G$ that contains a $v$, we can find the expected number of words as $\frac{|(\gamma,v)|}{|\gamma|}$ — $|\gamma|$ is the number of possible Euler cycles in $G$.

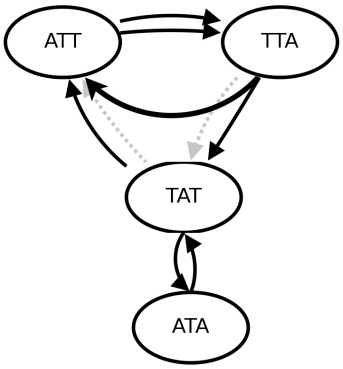


Figure 3.3: Replacing the path representing `TTATT` with a single edge

### How to evaluate

A very useful number to have for each word is $P_w$, the probability that a random word is at least as far from the expected value as $w$ is. This depends on how the expected/counted ratio is expected to be distributed. When we consider how words can overlap, this becomes somewhat costly to calculate[3], but it's possible to get good estimates by calculating a deviance residual for each word count. These are expected to be spread out in something close to a normal distribution, which is convenient to work with.

A side effect of the nonrandom nature of DNA is that while the shape of the spread of the deviance residuals is indeed close to a normal distribution, the standard deviance $\sigma$ is larger than the expected 1, and increases with the expected frequency of the word. This can be compensated for by using the actual word counts to create a model of expected variance ($\sigma^2$) as a function of the expected frequency; dividing each deviance residual by the calculated $\sigma$ normalises it to the expected range. After the normalisation, finding $P_w$ is trivial, since the standardised residuals should act like a normal distribution.

To indicate how statistically significant a given word count is, we calculate an E value. Given the length $n$ of the sequence, the expected count $x$, and $P_w$, the E value is $E = \frac{n}{x} \cdot P_w$. This is a weighted Bonferroni correction which de-emphasises words we expect, from the k-mer composition, to be rare (or conversely, it emphasises words we already expect to be common that still manages to stand out).

[3]But perfectly possible to do, see [Rod06].

# 4 Web application

## 4.1 Browse by genome

### 4.1.1 Summary

The first module of the web application makes it possible to look up data per genome — either by browsing a complete list of accession IDs and names, or by directly entering an ID. It is worth noting that each accession ID (and by extension, each entry) refers not to a complete genome, but to one chromosome or even plasmid. For the time being, this means that it's neccesary to look up several entries to get complete coverage of the genome of one organism. This might change in future development.

### 4.1.2 The table

Each entry is a link to a new page presenting the entry, mainly a table presenting the words in it we have data for; each row represents one word (see 4.1). The table is paged, displaying at most 100 entries at once. It is also sorted on the E-value taken over the entire genome, more on that in 4.1.3.

| Word | | All | | Coding | | Non-coding | | C/NC - ratio | | C/NC over-representation | | Position map |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ACCGCACT | 1574 | 74.21 | 1030 | 51.41 | 521 | 22.22 | 1.98 | $C_N$ | 0.85 | CN | |
| | AGTGCGGT | 0.0000 | 21.21 | 0.0000 | 20.04 | 0.0000 | 23.45 | | | | | |

Figure 4.1: One row, with data for one word

From the left, a complete row (e.g. 4.1) is constructed from these fields:

- A link to the pattern extension page, with this word as the seed (see 4.3).
- The word, in both reading directions.
- Three subtables, with the headings *All*, *Coding* and *Non-coding*.
- A number and image, labeled together as *C/NC-ratio*.
- Another number and image, labeled together as *C/NC - overrepresentation*.
- A longer barcode-like image with the header *Position map*.

### 4.1.3 Word counts

The statistics for each word are done over three subsets of the entire chromosome or plasmid: The first, which is also the one used for sorting purposes, covers the entire chromosome or plasmid. The second subset covers only those areas that are assumed to code for one or more genes, and the third subset only those not in the second. These correspond to the headings *All*, *Coding* and *Non-coding*. For each of these headings, each row contains a four cell sub-table. Clockwise from the top left, the cells are:

- How many times the word has been counted in this subset.
- How many times the word was expected to occur in this subset.
- The ratio of the previous two.
- The E value.

The $\frac{Counted}{Expected}$-ratio for the entire sequence ("All") determines the background colour of the entire row: If the word is less common than expected, the row is marked with a gray background, as opposed to the default white.

### 4.1.4 Ratios

Since we have separate word counts for the coding and noncoding regions of a sequence, a fairly obvious number to include was the ratio of these. The first number is simply $\frac{counted_{coding}}{counted_{noncoding}}$. In order to make the list easier to scan, and to visualize this number, a small image is generated from the same data. This image can be considered a bar chart of sorts, where the height of the letters C and N are proportional to the percentage of occurances of this word that were in the Coding and Non-coding subsets, respectively.

The second number is $\frac{R_{coding}}{R_{noncoding}}$, where $R = \frac{counted}{expected}$. In other words, how much more or less overrepresented the word is in coding compared to noncoding regions. The accompanying image works exactly like the previous one.

### 4.1.5 Position map

The last item in a row is an idealized map of where in the sequence the word is more or less common. Most obviously, this is a convenient way to estimate if two words occur together, but it also gives an indication of how common a word is, and how evenly spread throughout the sequence.

In the position map, the density is indicated on a scale from black to white, with blue indicating no occurences. The actual values in words/kBases for each colour are not fixed; the only constant is that white [1] indicates the highest density for that particular word in this particular sequence.

[1] RGB(255, 255, 255)

The map is generated by splitting the sequence into as many buckets as the width of the final image in pixels.[2] The number of words per bucket are then counted[3], and the highest word count is stored as $count_{max}$. The map starts with a blue background color, and each nonempty bucket is then rendered as a vertical line in the map. The colour is determined as RGB(i,i,i), where $i = 255 \cdot \frac{count}{count_{max}}$, rounded down to the nearest integer.

This non-fixed, normalised colour scale makes it harder to compare maps for different sequences or even different words in the same sequence. However, it solves the big problem with fixed values, namely that a max value high enough to not clip into an uniformly white map on the most common words would lead to a map using only a few values in the black end when dealing with less common words.

## 4.2 Export

Some of the tables can be exported in an XML or tab–separated format. At the moment, this is limited to the results of a search for a word or pattern, or the list of all words in a sequence.

The relevant links are at the bottom of the page, see figure 4.2.

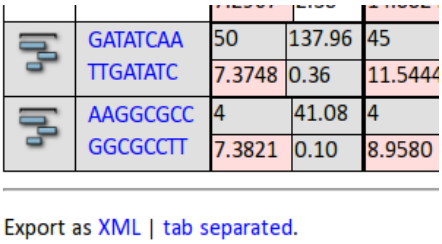


Figure 4.2: The export links.

In the first case, the XML format looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<wordlist>
  <line>
    <sequenceID>EX_AMPLE01</sequenceID>
    <word>ATGCATGC</word>
    <occCounted>100</occCounted>
    <occExpected>1</occExpected>
    <eVal>1e-10</eVal>
    <name>An example sequence.</name>
  </line>
</wordlist>
```

[2] In NC_007146, Haemofilus Influenzae, which has a fairly typical 1.9 MBases, each pixel and bucket represents about 4.8 KBases.

[3] A word is counted as being in the bucket it starts in.

And in the second case, like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<wordsInOne>
  <line>
    <word>ATGCATGC</word>
    <wordC>GCATGCAT</wordC>
    <cnRatio>2.0</cnRatio>
    <occRatio>2.0</occRatio>
    <all>
      <occCounted>30</occCounted>
      <occExpected>10</occExpected>
      <eVal>0.01</eVal>
      <ceRatio>3.0</ceRatio>
    </all>
    <coding>
      <occCounted>20</occCounted>
      <occExpected>5</occExpected>
      <eVal>1e-3</eVal>
      <ceRatio>4</ceRatio>
    </coding>
    <noncoding>
      <occCounted>10</occCounted>
      <occExpected>5</occExpected>
      <eVal>0.5</eVal>
      <ceRatio>2</ceRatio>
    </noncoding>
  </line>
</wordsInOne>
```

## 4.3 Pattern extension

### 4.3.1 Why

The word counts and all following statistics are done on words of a fixed length; currently 8 bases. However, there is no reason the overrepresented patterns in actual genomes should be exactly this length. This mismatch leads to one overrepresented pattern showing up as several words with similar characteristics. In the case of a pattern longer than the word length these will overlap quite well, while a shorter pattern gives lower-quality results corresponding to the different possible surrounding bases.

To help with the cases where the pattern is longer than one of the counted words, it is possible to use a word (indeed, any base sequence) as a seed to search for a common surrounding pattern.

### 4.3.2 How

#### The idea

The pattern expanding algorith takes a base sequence, and returns a list. Each list element contains a count of how many times each possible base has been observed at a given offset from the seed word, and a number indicating the statistical quality of this data.

#### The quality measure

The pseudocode in 4.1 is fairly straightforward, but the quality measure in *analyze()* in 4.2 deserves some further explanation. When reading a sequence, a count is kept for each base. When also considering the complementary reading, the A and T count will obviously be identical[4], as will G and C. From the combined AT and GC counts, we can trivially construct the overall $P_A = P_T = \frac{count_{AT}}{count_{AT+GC}}$ and $P_G = P_C = \frac{count_{GC}}{count_{AT+GC}}$.

Using the Kullback-Leibler divergence between the expected (P) and measured (Q) base ratios gives a useful measure of the information gain it represents.[Ren61].

$$D_{KL} = \sum_i Q_i \cdot log_2(\frac{Q_i}{P_i}) \qquad \text{where } i \in A, T, G, C \tag{4.1}$$

In a sequence with a completely even P=0.25 for all bases, the highest possible $D_{KL}$ occurs when only one base is observed: $1 \cdot log_2(\frac{1}{0.25}) = 2$. The other three terms will be $0 \cdot log_2(0)$ and thus undefined, but it is convenient to treat this as 0 for the sake of getting a useful result. With $P_{AT} \neq P_{GC}$ it will be higher, but even an unusual 30/70 skew would only gives a maximum at about $log_2(1/0.15) = 2.7$. The lower bound occurs when P=Q, and $D_{KL} = 4 \cdot \frac{1}{4} \cdot log_2(\frac{P}{P}) = 4 \cdot log_2(1) = 0$.

The cutoff rate for what is an acceptably good base, the *QualityLimit* in the pseudocode, has been arbitrarily set to the quite low 0.2 bits in the web application. This is high enough to give acceptably short matches, but low enough to give some leeway for the pattern generating code.

#### The pattern generator

The list of base counts and quality indicators can be used in itself, but both for further use and immediate legibility it is convenient to transform it into a text pattern. The process used takes a percentage limit as an additional argument beside the list of counts; the end results are presented for a range of values of it.

The basic idea is that a single base, or the top two together, have to make up at least the given percentage of the observations in a given position; otherwise it will be set to _, the marker for "any single base". Additionally, the top or top two bases have to have a certain margin above the highest of the ignored bases — this margin has been arbitrarily decided as 5% of the percentage limit.

[4]Since each A has a matching T on the opposite strand, and the reverse.

```
Given
    Seed, Sequence, QualityLimit, pAT, pGC
    MaxSkip=1

Do {
    PosList = searchFor(seed in sequence)
    Result = new List()

    #This loop expands onwards from the end of the seed.
    Skip=0
    Offset = stringLength(Seed)
    while(True) {
        Basecount = countBases(Sequence, PosList, Offset)
        Quality = analyze(Basecount)
        Offset ++
        if (Quality >= QualityLimit or Skip < MaxSkip) {
            Result.append(Basecount, Quality, Offset)
            if (Quality < QualityLimit) Skip++
            else Skip=0
        } else break
    }

    #This loop expands backwards from the start of the seed.
    Skip=0
    Offset=-1
    while(True) {
        Basecount = countBases(Sequence, PosList, Offset)
        Quality = analyze(Basecount)
        Offset --
        if (Quality >= QualityLimit or Skip < MaxSkip) {
            Result.append(Basecount, Quality, Offset)
            if (Quality < QualityLimit) Skip++
            else Skip=0
        } else break
    }
}
return Result
```

Table 4.1: Pattern expansion, pseudocode

*Functions*

```
function countBases(Sequence, PosList, Offset) {
    Basecount = new structure {
        bases = {A=0, T=0, G=0, C=0},
        total = 0
    }

    foreach Position in PositionList {
        Base = Sequence[Position + offset]
        Basecount.bases[Base] ++
        Basecount.total ++
    }

    return Basecount
}

function analyze(Basecount, pAT, pGC) {
    Bits = 0
    foreach Base,Count in Basecount.bases {
        Ratio = Count/Basecount.total
        if (Base=A or Base=T) {
            Bits = Bits + Ratio * log2(Ratio/pAT)
        else
            Bits = Bits + Ratio * log2(Ratio/pGC)
    }
    return Bits
}
```

Table 4.2: Helper functions for pattern expansion, pseudocode

An example might make the reasoning behind this margin limit clearer: Given a base count $\mathbf{B} \leftarrow \{A : 40\%, T : 30\%, C : 29\%, G : 1\%\}$ and a limit of 70%,. `A` alone doesn't qualify, but `A+T` add up to 70%. It would still seem wrong to represent this count as `[AT]` when only a tiny margin separates it from being `[AC]` — and indeed, with a margin of only 1 percentage point, it falls short of the 3.5 pp. margin limit and will be represented as `_` instead.[5]

The same example also highlights a weakness in this rather simple approach: Some cases that would be more precicely labeled "all bases except X" end up as "any base". This only represents one more accepted base in a given position (from 3 to 4), so the reduction in specificity is limited.

| Pattern symbols | |
|---|---|
| `A` | Exactly the base `A` |
| `[AT]` | Either `A` or `B` |
| `_` | Any single base |

### The presentation

The pattern generation described in the previous section is applied several times with different percentage limits, from 50%[6] to 90%.

The generated patterns for each setting are presented in a simple table (fig. 4.3). The different percentage settings have different uses — the very highest are useful for finding strongly conserved bases, while moving towards the lower end makes for more specific patterns that are more useful when searching and connecting.

| Pattern limit | Pattern |
|---|---|
| 50 | AAA[AT]TT[AG]**ACCGCACT**TT[TA] |
| 60 | AA[AT][AT][TA]T_**ACCGCACT**TT[TA] |
| 70 | [AT][AT][AT][AT][TA][TC]_**ACCGCACT**TT_ |
| 80 | [AT][AT][AT][AT][TA][TC]_**ACCGCACT**T__ |
| 90 | __[AT][AT]___**ACCGCACT**T__ |

Figure 4.3: Generated patterns at different percentage limits. The seed word was `ACCGCACT`.

For each of the percentage levels, an attempt is made to show which other known overrepresented words in the same sequence are likely to be involved in the same pattern. To do this, the generated pattern is split into sub-patterns of the same length as the already known words[7], and each of the generated patterns is looked up in the database.

[5] In order to get `[AT]`, `C` could be no higher than 26.5% — if this is a reasonable margin is of course debatable.

[6] Note that the closest distribution allowed at 50% is limited by the minimum separation of 2.5% — the tightest possible distribution for `[AT]` would be {A:26.25%, T:26.25%, G:23.75%, C:23.75% }, which would account for a perhaps not surprising 52.5%.

[7] Currently eight bases.

*Given*

Basecount, PercentageLimit

*Do*

MinSeparation $\leftarrow \frac{PercentageLimit}{20}$
Best $\leftarrow \frac{100 \cdot b.best.count}{b.total.count}$
BestTwo $\leftarrow \frac{100 \cdot (b.best.count + b.secondbest.count)}{b.total.count}$
Second $\leftarrow \frac{100 \cdot b.secondbest.count}{b.total.count}$
Third $\leftarrow \frac{100 \cdot b.thirdbest.count}{b.total.count}$

```
if Best > PercentageLimit:
    if (Best - Second) < MinSeparation:
        return "_"
    else:
        return b.best.base
if BestTwo > PercentageLimit:
    if (BestTwo - Third) < MinSeparation:
        return "_"
    else:
        return "["+b.best.base+b.secondbest.base+"]"

return "_"
}
```

Table 4.3: Pattern generation, pseudocode

Each row is then color coded according to the E-value of the word it represents — green for $< 0.1$, red for the rest. For an example, see 4.4. Each of the matching words are then presented with a bit more statistics.

**Overrepresented subwords, using the 6o-limit pattern:**

| A | A | [AT] | [AT] | [TA] | T | _ | A | C | C | G | C | A | C | T | T | T | [TA] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | A | T | T | G | A | C | C | G | | | | | | | |
| | | | T | T | T | G | A | C | C | G | | | | | | | |
| | | | A | A | T | G | A | C | C | G | | | | | | | |
| | | | T | T | T | A | A | C | C | G | | | | | | | |
| | | | | T | T | G | A | C | C | G | C | | | | | | |
| | | | | T | T | A | A | C | C | G | C | | | | | | |
| | | | | A | T | G | A | C | C | G | C | | | | | | |
| | | | | T | T | T | A | C | C | G | C | | | | | | |
| | | | | | T | G | A | C | C | G | C | A | | | | | |
| | | | | | T | A | A | C | C | G | C | A | | | | | |
| | | | | | T | T | A | C | C | G | C | A | | | | | |
| | | | | | T | C | A | C | C | G | C | A | | | | | |
| | | | | | | G | A | C | C | G | C | A | C | | | | |
| | | | | | | T | A | C | C | G | C | A | C | | | | |
| | | | | | | A | A | C | C | G | C | A | C | | | | |
| | | | | | | C | A | C | C | G | C | A | C | | | | |
| | | | | | | | A | C | C | G | C | A | C | T | | | |
| | | | | | | | | C | C | G | C | A | C | T | T | | |
| | | | | | | | | | C | G | C | A | C | T | T | T | |
| | | | | | | | | | | G | C | A | C | T | T | T | T |
| | | | | | | | | | | G | C | A | C | T | T | T | A |

Figure 4.4: A table matching a pattern to known overrepresented words

There are some cosmetic issues with the current presentation. It would make the table presenting the patterns easier to compare if the columns were lined up, and the tables matching patterns to words should filter out subpatterns with more than a certain number of expansions: The highest percentage numbers tend to result in long stretches of "any base acceptable", which leads to a very full but not very informative table. The same table could also stand some improvments as to what information is and isn't presented on each row.

## 4.4 Search by word or patttern

The other main approach to the website is to search for a word or pattern in all available genomes. The patterns used are the same as described on 4.3.2. Search results are presented in a simple table, sorted by E-value; there are links to the pages for the relevant genomes.

Searching for a pattern in all registred words (currently about 36.5 million) takes a

| GenID | Word | Counted | Expected | E | Name |
|---|---|---|---|---|---|
| NC_009566 | AAGTGCGG | 1669 | 89.45122 | 0.0 | Haemophilus influenzae PittEE, complete genome. |
| NC_000907 | AAGTGCGG | 1687 | 90.82277 | 0.0 | Haemophilus influenzae Rd KW20, complete genome. |
| NC_007146 | AAGTGCGG | 1732 | 94.63402 | 0.0 | Haemophilus influenzae 86-028NP, complete genome. |
| NC_008309 | AAGTGCGG | 1478 | 81.30756 | 0.0 | Haemophilus somnus 129PT, complete genome. |
| NC_009567 | AAGTGCGG | 1725 | 90.49306 | 0.0 | Haemophilus influenzae PittGG, complete genome. |
| NC_009655 | AAGTGCGG | 1857 | 125.1344 | 3.87e-250 | Actinobacillus succinogenes 130Z, complete genome. |
| NC_006300 | AAGTGCGG | 1622 | 120.8713 | 3.21e-223 | Mannheimia succiniciproducens MBEL55E, complete genome. |
| NC_002663 | AAGTGCGG | 1006 | 71.79501 | 3.85e-205 | Pasteurella multocida subsp. multocida str. Pm70, complete genome. |
| NC_008510 | ACGTGCGG | 123 | 26.31049 | 2.9e-14 | Leptospira borgpetersenii serovar Hardjo-bovis JB197 chromosome 1, |
| NC_008508 | ACGTGCGG | 121 | 26.21443 | 1.18e-13 | Leptospira borgpetersenii serovar Hardjo-bovis L550 chromosome 1, |
| NC_006369 | CCGCACAG | 96 | 21.7459 | 2.09e-12 | Legionella pneumophila str. Lens, complete genome. |
| NC_010175 | CCGCACTC | 573 | 170.9399 | 3.303e-08 | NO INFORMATION |
| NC_001851 | CCGCACCA | 18 | 1.292186 | 4.1e-05 | Borrelia burgdorferi B31 plasmid lp28-1, complete sequence. |
| NC_009342 | CCGCACAC | 235 | 101.8795 | 0.021723 | Corynebacterium glutamicum R, complete genome. |
| NC_010117 | CCGCACAA | 109 | 49.11047 | 0.173578 | NO INFORMATION |
| NC_009663 | CCGCACCG | 128 | 58.48295 | 0.213033 | Sulfurovum sp. NBC37-1, complete genome. |
| NC_010117 | CCGCACGG | 75 | 32.23784 | 0.40044 | NO INFORMATION |
| NC_008599 | AAGTGCGG | 76 | 27.52938 | 0.772444 | Campylobacter fetus subsp. fetus 82-40, complete genome. |
| NC_010278 | CCGCACCG | 412 | 224.4431 | 0.783901 | NO INFORMATION |
| NC_008741 | CCGCACGG | 96 | 35.07931 | 0.820431 | Desulfovibrio vulgaris subsp. vulgaris DP4 plasmid pDVUL01, |
| NC_002946 | CCGCACCC | 240 | 89.46938 | 0.850625 | Neisseria gonorrhoeae FA 1090, complete genome. |
| NC_009228 | CCGCACCC | 18 | 4.49909 | 0.924796 | Burkholderia vietnamiensis G4 plasmid pBVIE04, complete sequence. |

Figure 4.5: The results of a pattern search

significant amount of time, so while a search for an exact word will use the complete database, a search for a pattern will instead use a significantly smaller database of only words with an E-value under 1. This is a mere 40 000 words, but by far the most interesting ones. Practically speaking, this is a huge speedup: As an example, an arbitrary pattern (`%GTGCG[CG]`) took 158ms in the short table, but 113 seconds on the complete data set. Searches for exact words work better with the database indexes and don't show a comparable speedup, so they are done against the complete dataset.

The above optimization was done after testing on only one database system (PostgreSQL 8.2), but the large speedup[8] was the difference between the pattern search being useful or not on that particular setup. Future use with other database systems should re-evaluate this.

[8] I only did enough testing to make sure the effect was consistent, but as a rough estimate the time spent decreased by a factor of 500 to 1000.

# 5 Software used

## 5.1 History

### 5.1.1 Rewrite

The first version of the website part started off as a PHP project. At the time, the plan was to do all work on the data in separate programs on the server side, and have a thin layer of presentation on top of database tables. As things progressed, it was convenient or obvious to do more work in this presentation layer, an also to call on some of the standalone programs on every view of some of the pages (specifically, the pattern expansion system). While this did work, it had several issues. The largest was probably the inconvenience of writing any code that depended partially on existing PHP code and partially on standalone C programs.

While it would certainly be possible to rewrite and reorganize within the existing code, converting to another language seemed like it would be better in the long run.

## 5.2 Python

An ideal language for this project would have to be several things:

- Convenient to write smaller scripts in. Being able to pull out and/or work with data with a minimum of overhead can be very useful, both for testing and for answering specific questions.
- Suitable for writing a web application. While it is quite possible to do this entirely by hand in almost all languages, having a ready framework saves a lot of time and is likely to be both faster and safer.
- Easy to integrate with C, or fast enough to not require it.
- Common enough that it is likely to be supported on the relevant server platforms and into the future.

In the end, I settled for Python. It has a clean syntax, and aims to be easy to learn, read and maintain. Writing a web application in Python alone isn't ideal, but there are several good frameworks available — for more on that, see the next section on Django. Integrating Python and C has a certain learning curve, but the documentation is thorough and the end results good.

As for popularity, it's always hard to judge the size of a software project unless it's one of the very largest in its category. Still, Python is available as packages for all the

relevant OSes and sees steady development and use, so it shows no sign of disappearing — quite the opposite.

### 5.2.1 Python vs. PHP

As mentioned in the history section, the original code was in PHP. It might be useful to look at how they differ:

- Python is intended as a general-purpose scripting language, while PHP is specifically intended for dynamic web pages. Writing command-line scripts in PHP is quite possible, but Python has an edge in useful tools.
- Typing works differently. PHP will silently convert variables, while Python decides the type when a value is assigned, and requires explicit conversions. The former can be more convenient, but the latter catches more bugs, and can be easier to read later.[1]
- Python deals better with errors. The PHP parser can run into problems that cause it to silently die, which is annoying to debug. Python also has a more robust exception system: All problems will thrown an exception, and catching it will let the program continue. Unlike PHP, this means you can try importing a module and then gracefully print an error page if something went wrong — PHP has an annoying tendency to die quietly on some types of problems.
- Both languages have very good documentation, except for in one critical area: Integrating C and PHP was very poorly documented (or at the very least, finding good documentation was hard) at the time I was looking for it. In contrast, Python has perfectly decent documentation for this.
- Organising code is a bit cleaner in Python. The module system makes it easy to convert a directory of python files into a module, and also keeps the global namespace less cluttered: In PHP, all functions in an imported file are available globally. Python allows this (`from module import *`), but also supports a more fine-grained version (`from module import aFunction`), and a namespace version (`import module`, and then `module.aFunction()` ). This is especially useful when importing modules that themselves use other modules. Note that PHP has started work on namespace support recently.

## 5.3 Django

Developing web applications in Python can be done with only what comes in the standard library, but using a ready-made framework has several benefits. The most obvious one is that it saves a good bit of work to use a ready-made system for handling requests, wrapping database access, and formatting output. Less obvious is how a good framework

[1] One of the guiding principles of Python is that *explicit is better than implicit*, for these reasons.

will mold the code produced with it into a better shape, by making a good structure the default and easiest way to write code.

Django is a python framework for writing database-backed webapps. It contains an ORM[2] system, a dispatcher for calling different functions depending on the URL, and a template system for writing the web pages separately from the python code.

### 5.3.1 ORM

The ORM layer is very useful not just for the website, but for any code using the database. It abstracts the database tables into a set of classes and objects, where the objects are linked together to match the tables. It's also possible to write custom methods in these classes, which is how the denormalized word/genome tables are glued together.

It also allows queries like "what are the coding/noncoding ratios for all words in a genome where the name contains "influenzae"" to be put together easily: The wordGenome class has a method for getting that ratio, so it's merely a question of filtering out the right words and using the right fields and methods:

```
for w in wordGenomeBestof.objects.filter(gen_id__name__contains="influenzae"):
    print w.gen_id.gen_id, w.word, w.c_nc_ratio()
```

A similar SQL query would be a bit more involved, partially because of the table layout, and partially because of the need for handling cases where the fields are missing or the divisor is zero. A query without this handling might look like this:

```
select g.gen_id, w.word, wgc.occ_counted/wgnc.occ_counted
from ((genomes g join word_genome_bestof w using (gen_id) )
    join word_genome_coding wgc using (gen_id) )
    join word_genome_noncoding wgnc using (gen_id)
WHERE g.name LIKE '%influenzae%';
```

The python code in the wordGenome class handles the assorted special cases, and does things in a rather more explicit way:

```
def coding(self):
    res = wordGenomeCoding.objects.filter(gen_id__exact=self.gen_id.gen_id)
    res = res.filter(word__exact=self.word)
    if not res is None and len(res) > 0:
        return res[0]
    else:
        return None

def noncoding(self):
    res = wordGenomeNonCoding.objects.filter(gen_id__exact=self.gen_id.gen_id)
```

[2]Object Relational Mapping

```
    res = res.filter(word__exact=self.word)
    if not res is None and len(res) > 0:
        return res[0]
    else:
        return None

def c_nc_ratio(self):
    nc = self.noncoding()
    c = self.coding()
    if c is None or nc is None:
        return 0
    if (nc.occ_counted == 0):
        return c.occ_counted
    else:
        return float(c.occ_counted)/float(nc.occ_counted)
```

### 5.3.2 Templates

In order to separate the underlying code from the presentation, Django has a template system for constructing the actual web pages. A template is simply a HTML file with extra formatting codes; templates can extend and include other templates.

## 5.4 Database system

The choise of RDBMS[3] was not a big issue: The two big free/opensource alternatives, MySQL and PostgreSQL are both good enough. I ended up using PostgreSQL since I was already familiar with it.

There is rather a lot of benchmarks and statements in support of either one, with both sides claiming victory in whatever field they hold more important. Referring to any of these wouldn't be very enlightening, so I will merely say that my experience with PostgreSQL during development has been entirely free of surprises.

## 5.5 Other software

In addition to the Python and the RDBMS, it takes a few extra pieces of software to go from source code to a working website.

### 5.5.1 Apache

The most obvious part of the equation is the webserver. While Django offers a development web server written in Python, this is solely intended for testing, since it does not

[3]Relational DataBase Management System — that is, the database software.

perform especially well. [4] The solution is to run a separate web server and make it call the Python code somehow.

When it comes to chosing a web server, it's hard to overlook Apache. It's the most used web server in the world, and exists as free, opensource ports for all relevant operating systems (and indeed a lot of irrelevant ones). The newest stable version as of writing is 2.2, and that's what I have used.

There are several ways to run Python code from Apache. The Django project recommends `mod_python`, but also supports methods like `FastCGI`.With `mod_python`, a python interpreter is embedded into each relevant server thread. The source code is compiled if neccesary[5] and loaded, and then stays in memory as long as the server thread lasts. This saves time, though at the cost of requiring a server restart whenever the code changes.

`FastCGI` splits the serving into two parts: The web server, and a separate process that runs the python code. These two parts communicate over an UNIX socket or TCP. This gives a bit extra overhead and requires more setup, but makes it possible to restart the Python server separately from the web server, and indeed to run these on different computers.

As of now the one installation of the project uses `mod_python`, but there is nothing tying it to any specific architecture in this regard.

### 5.5.2 The rest

Django allows trading memory for CPU time by caching pages in memory. If an URL matches one used earlier it will be pulled from the cache instead of generated again, saving time and resources.

To store the cache, Django supports rather a lot of different backends. The most interesting one is perhaps a separate daemon called Memcached. This isn't a Django or Python project, but a stand-alone program that can be used from any other application. [6] While it can run on the same computer as the web server, it can also be set up on one or more separate computers. There are several recommended tricks to make the most out of memcached, like running instances of it on CPU-bound servers with low memory usage — this is however way beyond my scope here.

Of course, Memcached isn't required to run anything, but if it's available it can certainly be used. Enabling it requires changing the `CACHE_BACKEND` in the Django configuration file.

The C code does obviously require a C compiler. I have only tested gcc 4.2, but any other modern compiler should be acceptable.

As of today the code is developed on FreeBSD. It should work on Linux[7], but compiling and running on Windows is likely to take some adaption.

[4] This is partially intentional. Being able to change the source code and immediately reload a page to see the effect is useful, but hard to combine with some forms of cache.

[5] Python stores precompiled code alongside the source code, in .pyc-files, to avoid compiling anything more than once.

[6] A side effect is that the cache persists through apache restarts, making it near enough impossible to test code changes while using it.

[7] It has been sporadically tested on debian/PA-RISC

# 6 The software tools

## 6.1 In short

The backend code for the web site is intended to allow users to write scripts to find more specific information and statistics. While parts are written in C for performance reasons, everything has a python interface to allow for more convenient scripting.

It might be useful to shortly explain C and Python before going in-depth. Both are programming languages, but have quite different style and uses. C is a common language with a long history that can produce very fast results, at the cost of being somewhat primitive and arcane. Python is a modern scripting language with a much easier syntax and style than C, though at the cost of being slower. Luckily, writing C code that can be used from Python is a well documented process, and allows a style where most of an application is in Python, but the most time-consuming parts are done in C.

## 6.2 An example

The best way to show what can be done is a worked example. Let's say we want to plot E-values versus the coding/noncoding ratio for a large set of words. The candidate tables are either the All-table, which contains every single word, or the BestOf-table, the very much shorter list of words with an E-value $< 1$. The latter is both the quickest to work with and contains the most interesting words, so we will use that.

As for presenting the data, merely dividing the coding/noncoding counts has a few issues. Firstly, it will fail if the noncoding count is zero, and secondly, it will compress any result where the noncoding count is larger than the coding into a narrow band $y \in [0, 1\rangle$, as opposed to the $y \in \langle 1, \infty]$ spread in the opposite case. One possible solution which produces a more easily readable spread of data $y \in [-1, 1]$ is $y = \frac{C-NC}{C+NC}$.

When the ratio function is done, producing the data is just a question of iterating through all the relevant words, extracting the data, and writing it to a temporary file. The actual code is table 6.1.

The specifics will be explained in the following sections and to some degree in the API. For now, let's continue to the plotting. The most convenient tool for producing plots from this kind of data is probably *gnuplot*, a standalone program that is usually installed on linux workstations. Given the data file produced by the example code[1], merely starting gnuplot and typing `plot "tmp_data", 0` will produce a scatter plot from the file, and draw a line in a contrasting color through the x axis. The result is a window like figure 6.1. Applying some heavyhanded smoothing to the data by fitting

[1] Two values per line, separated by whitespace.

```
from wordsInOne.models import *

def ratio(c, nc):
   if (c is None or nc is None):
      return 0.0
   sum = c.occ_counted + nc.occ_counted
   diff = c.occ_counted - nc.occ_counted
   if sum == 0:
      return 0.0
   else:
      return float(diff)/sum

file = open("tmp_data", "w")

for word in wordGenomeBestof.objects.all():
   if word.occ_counted == 0:
      continue
   nc = word.noncoding()
   c = word.coding()
   file.write("%f %f\n" % (word.e_val, ratio(c, nc)))
```

Table 6.1: Example code for extracting data

it to a bezier curve with as many control points as there are data points can be done with the command `plot "tmp_data" smooth bezier, 0`. The resulting plot looks like figure 6.2. The huge dip on the right end seems to be an anomaly (the last few data points happen to be negative, that end of the graph is sparsely populated, and the curve fitting reacts strongly to the combination of those factors). The smaller dip on the left side is however harder to ignore, especially since it happens in the area of the graph with the largest amount of data points.

Gnuplot also has another smoothing mode that is best described by quoting their manual:

> The acsplines option approximates the data with a "natural smoothing spline". After the data are made monotonic in x, a curve is piecewise constructed from segments of cubic polynomials whose coefficients are found by the weighting the data points; the weights are taken from the third column in the data file. (...)
>
> Qualitatively, the absolute magnitude of the weights determines the number of segments used to construct the curve. If the weights are large, the effect of each datum is large and the curve approaches that produced by connecting consecutive points with natural cubic splines. If the weights are small, the curve is composed of fewer segments and thus is smoother; the limiting case is the single segment produced by a weighted linear least squares fit to all the data. The smoothing weight can be expressed in terms of errors as a statistical weight for a point divided by a "smoothing factor" for the curve so that (standard) errors in the file can be used as smoothing weights.

Using that requires another column in the data file. In this case, we will use the total word count as the weight, which is fairly easy to add: All it takes is to replace the line that writes to the file with
`file.write("%f %f %d\n" % (word.e_val, ratio(c, nc), word.occ_counted))` and running the script again. The resulting plot is figure 6.3.

As for what this means, well. There does seem to be a weak correlation between the very lowest E-values and a higher number of occurances in non-coding areas. A few plots with haphazardly applied smoothing is far from strong evidence, but it could perhaps be worth doing a more thorough analysis — and getting the raw data for that isn't hard.

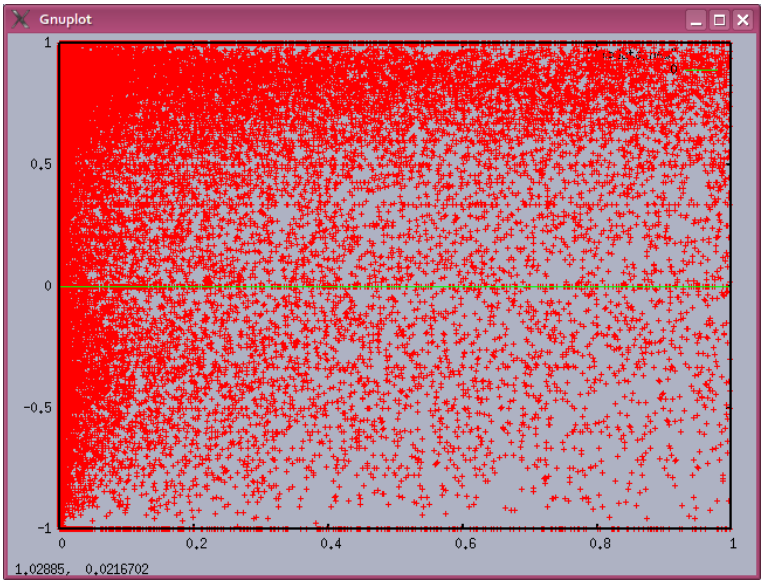


Figure 6.1: Scatter plot

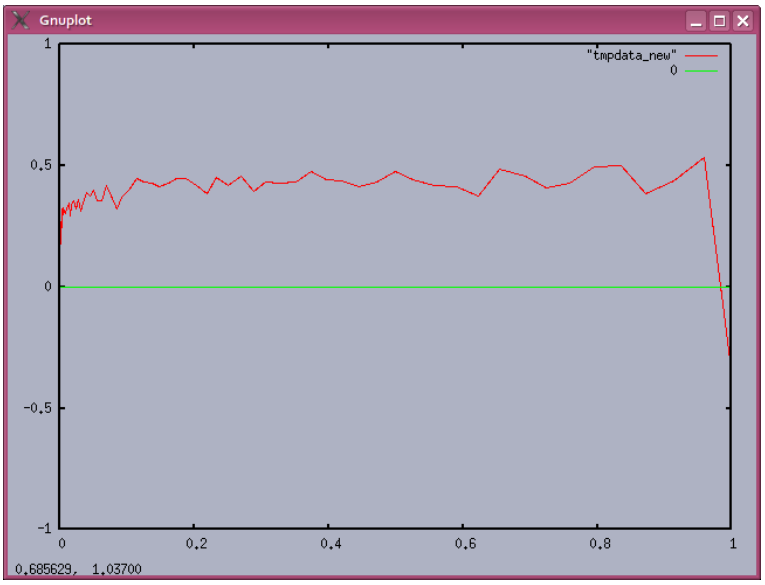


Figure 6.2: Bezier smoothing

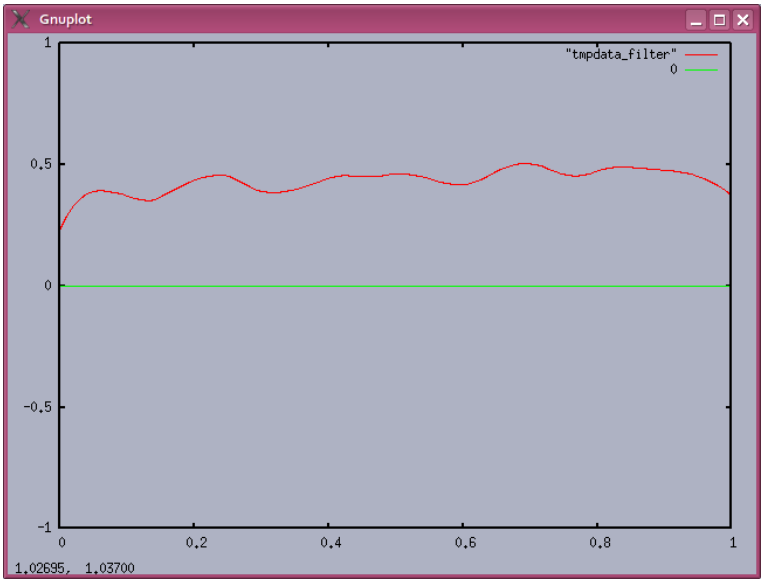


Figure 6.3: Acsplines smoothing

## 6.3 A more complex example

### 6.3.1 Idea

The previous example is a simple case of retrieving some data from the database. In order to show off some less obvious opportunities, the next example will expand all words with $E < 1$ in a sequence into patterns, and then give a list of all the possible pairings of patterns, with the best alignment, and a score for how well they overlap — sorted by the score.

The basic structure of the script is fairly simple:

```
For each word in (all overrepresented words in a sequence):
    Expand a pattern using the word as the seed
    Compare this pattern to all previously found patterns

Sort and print the results
```

### 6.3.2 Code

To be able to import the modules that use django code (which is most of them) in a standalone script, it's neccesary to add the directory con taining the project to `sys.path`, and to set `DJANGO_SETTINGS_MODULE` appropriately.

```
#!/usr/bin/env python

# First, set the environment up so we can import django modules
import sys
import os
sys.path.append('../')
os.environ['DJANGO_SETTINGS_MODULE'] = 'bio.settings'

# Then import the other modules
from django.conf import settings
from wordsInOne.models import *
from py_helper.models import *
from py_helper import *
from operator import *
```

Looking at the pseudocode, we first need to get a list of all the overrepresented words in a given sequence. Since we want to use this script as a command line tool, the sequence ID is a command line argument. The other command line argument is the pattern percentage limit (see 4.3.2), though it is optional and has a default value. We also need to read the raw sequence data so the pattern finding code can work on it.

```
# Check that we have gotten at least one argument
if len(sys.argv) < 2:
    print "Use ./cluster.py SequenceID [pattern percentage imit]"
    print "e.g. ./cluster.py NC_12345678 70"
    exit()
seqID = sys.argv[1]

# Get the percentage limit, if given; default to 70
plimit = 70
if len(sys.argv) >= 3:
    plimit = int(sys.argv[2])

# Read the sequence data
(data, datalen, AT, GC) = readSequence(seqID)
words = wordGenomeBestof.objects.filter(gen_id = seqID)
```

The actual process of looking for a possible pattern around a seed word happens in several stages. The first step (`find_positions`) is to search for the seed word in the sequence data. This gives us two lists; one per reading direction. These two lists can then be used to count the bases at different offsets (`basecount`): This gives us two new lists. These two lists are the base counts in the potentially interesting positions before and after the seed words, respectively.

To work with the fairly raw data in the position lists, they are wrapped in two layers. The first turns each position in the list into an object with a few more useful functions (`base_wrap`), and the second wraps these objects plus the seed word to provide an overall view of the possible pattern they represent (`ExpandResult`).

```
patterns=[]
scorelist = []
i=0

for w in words:
    word = w.word
    # Search for the word
    (pl, plC, plCount, plCCount) = find_positions(data=data, word=word)

    # Count bases before/after the word
    bPre, bPost = basecount(data, word, pl, plC, 0.2)

    # Wrap the raw data up in convenience classes
    basesBefore, basesAfter = base_wrap(bPre), base_wrap(bPost)
    er = ExpandResult(basesBefore, basesAfter, word, plimit, seqID)

    # Match the freshly constructed pattern with
    # the ones found in previous iterations
    addNew(patterns, scorelist, er, i)
    i += 1
```

The previous code will give us a list of `ExpandResult`-objects, one for each seed word. In order to find which of these have the best overlaps with each other, it's neccesary to compare each one with all the others. In this case, each new pattern is compared to all members in a list of previously constructed ones, and then added to the end of it; the results are stored in a separate list with enough information to reconstruct the best match they refer to.[2]

The `addNew`-function adds `er` (the new pattern) to `patterns`, and the scores for the best overlaps compared to the patterns already in it are added to `scorelist`

Finding the best overlap between two patterns can be done in a fairly simple manner: Try all possible overlaps, score each, and return the one with the best score. In the code, the two patterns are referred to as A and B. Pattern A is expanded with filler characters on each end to simplify the scoring code — if we use _ to designate the filler, it looks like this:

```
       ___AAAA___
Start: BBBB
End:         BBBB
```

[2]That is: Which two patterns, their relative offset, and if the reading directions were identical or complementary.

By doing the scoring in another function, this ends up as quite simple and compact code. One python idiom might be worth explaining: Sorting is in-place, and `scores[-1]` means "the last element in `scores`".

```
def bestOverlap(pA,pB, complement=False):
    lA = len(pA)
    lB = len(pB)

    # Fill out the top pattern with spacers before/after
    filler = ['_'] * (lB-1)
    pA = filler + pA + filler

    # Try all overlapping offsets
    scores=[]
    for offset in range(lA+lB-1):
        scores.append( (overlapScore(pA, pB, offset), 1+offset-lB ))
    scores.sort()
    return scores[-1]
```

The actual scoring is a bit arbitrary, as benefits a proof-of-concept script.

```
_score_1to1 = 1 # A vs A
_score_2toSame = 1 # [AT] vs [AT]
_score_1to2 = 0.5 # A vs [AT]
_score_2toHalf = 0.25 # [AT] vs [AG]
_score_Miss = 0 # No match, or one side is _

def overlapScore(a,b, offset):
    score=0.0
    for pos in range(len(b)):
        bA = a[offset+pos]
        bB = b[pos]

        if bA == '_' or bB == '_':
            score += _score_Miss
            continue

        if bA in 'ATGC':
            if bB == bA:
                # A vs A
                score += _score_1to1
                continue
            if bB[0] == '[':
                if bA in bB:
                    # A vs [AT]
                    score += _score_1to2
                    continue
```

```
            # A vs [GC]
            score += _score_Miss
            continue

        # bA must be of the [XY]-type if we get here
        if bB[0] == '[':
            if bA[1] in bB and bA[2] in bB:
                # [AT] vs [AT]
                score += _score_2toSame
                continue
            if bA[1] in bB or bA[2] in bB:
                # [AT] vs [AC]
                score += _score_2toHalf
                continue
            # [AT] vs [GC]
            score += _score_Miss
            continue


        # bB must be a single base
        if bB in bA:
            # [AT] vs A
            score += _score_1to2
            continue

        #[AT] vs C
        score += _score_Miss
    return score
```

The end result of this is the `patterns`-list, and the `scorelist`. Printing both to standard output in a whitespace-separated format is simple and makes it easy enough to work with it later:

```
# Print the expanded patterns
i=0
for p in patterns:
    pt = reduce(add, p.pattern())
    print i, p.word, pt, rev_word(pt)
    i += 1

print "###"

# Print the matches and scores, sorted
scorelist.sort( )
for s in scorelist:
    score, offset, direction, i, j = s
    print score, offset, i, j,
    if direction:
```

```
        print '!'
    else:
        print '='
```

The python `reduce` function applies an operation (in this case, addition) to reduce an array into one object: $reduce(*, (x_1, x_2, x_3, x_4)) = (((x_1 * x_2) * x_3) * x_4)$. Addition on python text objects concatenates them, so this turns an array of short strings (one per position) into a single long string representing the pattern.

The output looks like this:

```
0 AAGTGCGG AAAGTGCGGT_[AG][AT][TA][TA][TA][TA]_ _[TA][TA][TA][TA][AT][CT]_ACCGCACTTT
1 ACCGCACT [AT][AT][AT][AT][TA][TC]_ACCGCACTTT_ _AAAGTGCGGT_[GA][TA][AT][AT][AT][AT]
2 AAAGTGCG _AAAGTGCGGT_[AG][AT][AT][TA][TA][TA]_ _[TA][TA][TA][AT][AT][CT]_ACCGCACTTT_
.
.
.
59 CTACCGCA [AT][AT][TA][TA]_CTACCGCA_[TC][TA]__ __[TA][GA]_TGCGGTAG_[TA][TA][AT][AT]
###
2.0 0 50 40 =
2.0 1 45 40 =
.
.
.
16.0 1 5 2 =
16.5 0 10 6 =
16.5 0 44 7 =
```

The first half is the patterns: The columns are id, seed word, pattern, pattern (complementary reading). The second half is the matches: The columns are score, offset, id of first pattern, id of second pattern, and relative reading directions.

The match in the last line means "pattern 44 and pattern 7 had their best alignment with the first base of pattern 7 0 bases to the left of the first base of pattern 44; the score was 16.5, and they used the same reading direction". This corresponds to this:

```
44 TCACCGCA [AT][AT][AT][TA][TA] T  CACCGCACTT[TA]__
7  CACCGCAC [AT][AT][AT][TA][TA][TC]CACCGCACTT[TA]_
```

And indeed, they seem to overlap nicely.

### 6.3.3 Uses

With the data from the above, it wouldn't be especially complicated to do further clustering, either by constructing new patterns from the existing pairs and then repeating the process on those, or by using the scores to gather closely connected groups more directly. This would also make for a useful future addition to the pattern expansion function of the website.

# 7 Database layout

The raw data from the word counting and statistics get stored in a small set of database tables, and the mappings of sequence IDs to human-readable names get another table.

The theoretically ideal database layout would be a normalized version like figure 7.1. 7.1.

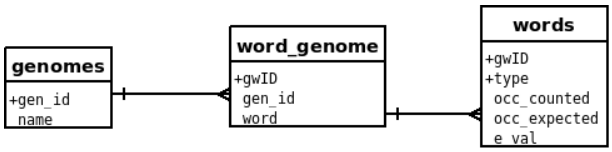


Figure 7.1: The theoretically ideal database layout

The `words` table would be huge — as of writing this, 109 046 293 rows. This isn't an issue in most situations, but any query involving a scan or a large join could take seriously long time. As an example, selecting all rows belonging to a single sequence takes about 20 minutes:

```
SELECT * FROM word_genome JOIN words USING (wgID) WHERE gen_id='NC_009566';
Time: 1230332.770 ms
```

Clearly, that would be inconvenient to wait for in the web interface. In order to speed this up, one option is to trade disk space and elegance for speed by denormalizing the relevant tables. This basically means joining the `word_genome` and `words` tables and storing the result as a single table. Incidentally, this is how the raw data from the statistics is output — this saves several hours when importing the data.[1] In addition to this, I have split the resulting table into three, one per each of coding, non-coding and all (see figure 7.3) This is more of a convenience (it saves merging the raw data) than a performance choice, though it does make queries that only want one of the word counts very slightly faster.

Even with that, though, searching for words matching a pattern is still way too slow for practical use when done against one of these three tables. To speed that up significantly, searches are only done against another smaller table containing only those word counts covering an entire sequence that have an E-value $< 1$.

[1] While importing the data doesn't happen very often, having it take twelve hours is still impractical.

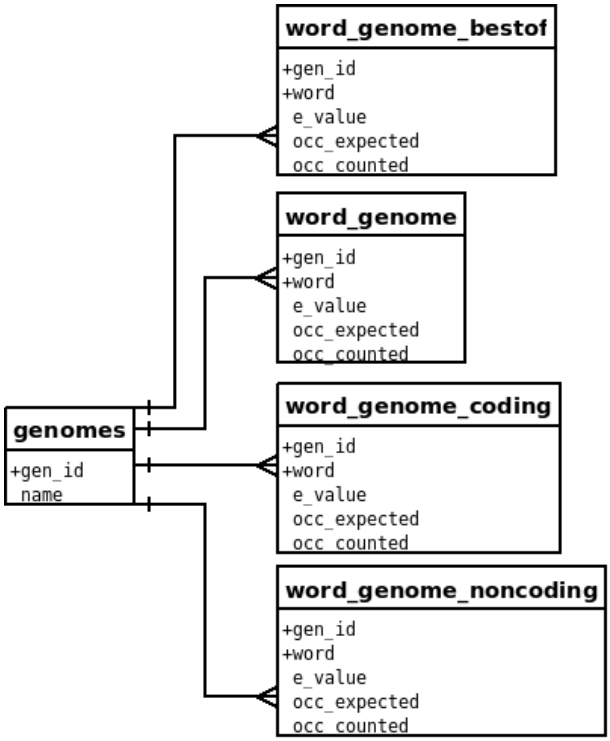


Figure 7.2: The actual database layout

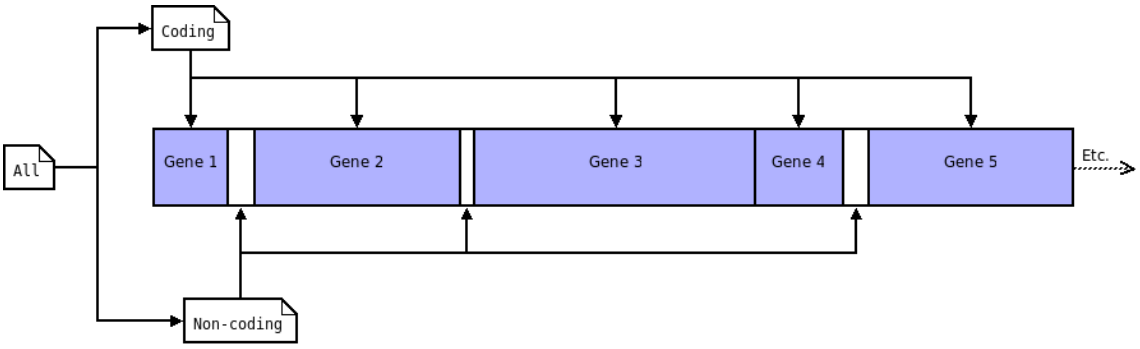


Figure 7.3: Coding, Non-coding and All

# 8 What has not been done

No programming or research project is ever completely done. In this one, there are a number of features and theories I would have liked to add and test, but that will have to wait until I can find the time.

## 8.1 In the code

In the pattern expanding code, the quality evaluation of the possible pattern is done per-base. Changing this to look at the k-mer count and probability should make it more specific, and would fit better with the approach used otherwise in the statistics.

The handling of patterns is arguably primitive. Using the basic idea from example 2 with some database backing to store a list of suspected motifs for each sequence would be quite interesting.

## 8.2 In the web application

The pattern expanding page doesn't handle well-preserved patterns above a certain length well — the nonlinear growth in CPU time and memory use in some pathological cases[1] can lead to a response time of many hours. Fixing this would require rewriting moderate parts of both the underlying C code and the page-generating python code; the current form prioritises clarity over efficiency.

Two of the graphical elements that were in the PHP version haven't yet been translated over: A letter map presenting the base frequencies in the pattern expander, and a circular position map in the same.

## 8.3 In general

One hypothesised way of finding functional words that we were hoping to test is to look for words occuring a certain distance from the start (or end) of known genes. While the relevant infrastructure is mostly in place now, there hasn't been time to connect the parts. Specifically, this would require reading the known gene positions from somewhere, and then for each word we want to check (e.g. every word we have a nonzero count for) use the word search code to get a position list, and look for correlations between that list and the gene positions.

[1] A good example is the insertion sequence in Shigella Dysenteriae, a 700+ bases long sequence in several hundred identical copies

A framework for handling the relations between different sequences would be very useful. Most importantly it would allow grouping different chromosomes and plasmids together as one genome, but it could also be used to group strains and families of bacteria together.

# 9 API

## 9.1 bio

### 9.1.1 bio.settings

The configuration file, `settings.py` does not surprisingly contain some settings used throughout the code. It has been adequately commented, so I won't reproduce it here.

## 9.2 bio.graphics

### 9.2.1 functions

`shiftUp(v1, v2)`: Given two numbers $v1$ and $v2$, scale them so $v1 > 10$, $v2 > 10$ and $v1/v2$ remains roughly the same.

### 9.2.2 Class lettermap

An object that returns a letter map: An image with two characters scaled to represent the ratio of two numbers. If both numbers are 0, a third character is displayed instead.

`__init__(val1, val2, letters=(u'C', u'N', u'-'))` : Creates the object. `val1` and `val2` are the values to be represented, `letters` is an array of the characters to be used to represent them — the third character is the one used if both values are 0.

`get()` Returns the URL of an image representing this image. If it doesn't already exist, it will be created and stored first by calling the `draw()` method.

`draw()` Creates the image, and stores it in the `MEDIA_ROOT` folder, as specified in the configuration file.

### 9.2.3 Class positionplot

An illustration of where in a sequence a given word or pattern is found.

`__init__(gen_id, word, data=None, datalen=None)` : Creates an object representing where `word` is in the sequence identified by `gen_id`. If both `data` and `datalen` are given, `data` is taken to contain the base sequence and `datalen` is its length; if either is missing the data will be read from disk when needed.

`get_linear()` : Returns the URL of an image representing this map. The image will be square; the leftmost edge represents the first base in the sequence. In the case of a circular chromosome or plasmid this is often but always the replication origin. The size is set by `PPLOT_LINEAR_SIZE` in the config file.

~~`get_circular`~~ : Like `get_linear`, but drawn as a circular genome. Not yet implemented, but `PPLOT_CIRCULAR_SIZE` in the config file is set aside to give the size.

## 9.3 bio.py_helper

Assorted utility functions, including the wrapper around the C code.

### 9.3.1 functions

`readfna(filename)` : Reads a text file in .fna format representing a base sequence. Returns `(data, AT, GC)` , where `data` is a string containing the raw base sequence, and `AT` and `GC` the AT and GC count, respectively.

`rev_word(word)` : Returns the complementary reading of the word or pattern in `word`.

`find_positions(data, word)` : Searches for `word` and its complementary reading in the sequence given in `data`. Returns `(pl, pl_rev)`, which are linked lists of positions for the given and complementary reading, respectively. They are `positionList`-objects, so see also that class.

`basecount(data, word, pl, pl_rev, q)` : Uses the given position lists (`pl` and `pl_rev`) to count the bases on each side of `word` in `data`. Continues outward in each direction until the "signal quality" is lower than `q`. Returns `(bases_rev, bases)`, linked lists of the base counts before and after the word, respectively. They are `base_py`-objects, see also that class.

`merge_positions(pl1, pl2)` : Merge two position lists, returning the header of the new list. Note that the original order is lost, since this works by changing the "next"-pointer in each object.

### 9.3.2 Class base_py

A count of how many times each base has been seen in one relative position. This class is iterable: Each `base_py` object knows which one is the next one in the relevant reading direction (that is — away from the seed word).

#### Properties

`pos` : The relative position. Negative when in front of the word, positive when after. The first position in front of the word is -1, while the first one after is L, where L is the number of bases in the word.

`quality` : How many percent of the total count is made up of the two most common bases. This is an integer, in the $[50, 100]$ range.

`ent_q` : The statistical quality of the "signal" in this position; See 4.1.

`count` : How many bases have been counted overall in this relative position.

`best` : Which base is the most common.

`bases` A dictionary where the key is a base and the value the count.

### 9.3.3 Class positionList

A list of positions. In a sequence of x bases, 0 is the first and x-1 the last valid position. `positionList` objects have just two properties, `pos` and `next`, and are iterable.

### 9.3.4 models

Some data-wrapping.

#### Functions

`base_wrap(bases)` : Wraps `base_py`-objects in `Base`-objects. Note that since `base_py` is a linked list, it suffices to use the first one in a chain as the argument.

`readSequence(seqID)` : Reads the sequence identified by `seqID`. This is a thin wrapper around `readfna()` that adds the file type and path; the path is set by `GENOME_DIR` in the config file.

#### Class Base

A wrapper around a `base_py`-object that can represent it as a pattern fragment, and contains a few convenience properties.

#### Methods

`__init__(b)` : Creates an object wrapping the `base_py`-object `b`.

`toPattern(limit=70)` : Returns a pattern fragment (that is, `X`, `[XY]`, or `_`) representing this base count. The way the limit works is described in 4.3.2. Note that `__str__` calls this function, so using a `Base`-object as a string is identical to using the return value of this function.

### Properties

`b` : The underlying `base_py`-object.

`bases` : The bases, sorted highest-count-first.

`values` : The base counts, sorted highest-count-first.

`percentages` : The base counts as percentage of the total, sorted highest-count-first.

### Class ExpandResult

A wrapper around a list (or two) of `Base`-objects and a word, that can represent the whole as a pattern.

### Methods

`__init__(bases_before, bases_after, word, percentage, gen_id=None)` : Creates a new object wrapping the word `word`, the base counts before and after it (`bases_before` and `bases_after`), with `percentage` as the percentage limit. The `gen_id` is the sequence ID this is from, and is required to use the `get_common_subwords` and `to_table` methods.

`pattern_before()`

`pattern_after()` : Returns the pattern describing the bases before or after the seed word.

`pattern()` : Returns a pattern describing the entire possible motif. This is the same as `pattern_before() + word + pattern_after()`.

`get_subwords(length=8)` : Returns an array of each subsection of the pattern with the given length. That is: If the complete pattern was `"AB[CD]E"` and the length was 3, it would return `("AB[CD]", "B[CD]E")`.

`get_common_subwords(length=8)` : Uses the subwords from `get_subwords` to look for over- or under-represented words in the table of known counts. The results are a list of tuples, each tuple on the form `(pattern fragment, result)`, where the `result` is a Django ORM query object — see `bio.wordsInOne.models`.

`to_table()` : Creates an XHTML table lining up the matches from `get_common_subwords()` with the pattern from `pattern()`. Note that this function might get moved somewhere more appropriate in a later revision.

### 9.3.5 Template tools

The `py_helper`-module also contains a tag and a filter for use when writing web page templates:

The tag is `menu`, and produces a series of links up in the page hierarchy from the current position.

The filter is `complement`, and returns the complementary reading of the word or pattern it's used on.

## 9.4 bio.wordInAll

The only code in this module is the views for the relevant web pages.

## 9.5 bio.wordInOne

For historical reasons, the models encapsulating the database tables are in this module. They are likely to be moved out to a generic helper module later, possibly `py_helper`.

### 9.5.1 models

Looking at the database layout, there are a number of tables but only really two *types* of them. Reflecting this, the different wordGenome tables all extend an abstract base class called `wordGenomeBase`. The concrete classes are `wordGenomeAll, wordGenomeCoding, wordGenomeNonCoding` and `wordGenomeBestof`.

#### Class wordGenomeBase

**Properties** The same as in the database table: `gen_id, k_order, word, occ_counted, occ_expected, std_residual, e_val`. Note that `gen_id` isn't a text field, but a `Genome` object — reading the text value of the field requires using `object.gen_id.gen_id`.

#### Methods

`complementary()` : Returns the complementary reading of the word.

`ratio()` : The counted/expected occurances of this word.

`c_nc_ratio()` : The coding/non-coding occurances of this word.

`c_nc_occ_ratio()` : Effectively `this.coding().ratio() / this.noncoding().ratio()`.

`e_class()` : A text description of the E value of this word: One of "e_low": $[0, 0.05>$, "e_medium": $[0.05, 0.1>$ or "e_high": $[0.1, \infty>$.

`uncommon()` : True if this word is less common than expected; False otherwise.

`coding()` : Returns an object representing the statistics of this word in the non-coding parts of the sequence.

`noncoding()` : Returns an object representing the statistics of this word in the coding parts of the genome.

### Class Genome

Just two properties, representing the database columns: `gen_id` and `name`.

# Bibliography


[AST+84] WL Albritton, JK Setlow, M Thomas, F Sottnek, and AG. Steigerwalt, *Heterospecific transformation in the genus haemophilus.*, Molecular Genetics and Genomics (1984), 358–363.

[Fit83] W. M. Fitch, *Random sequences.*, J Mol Biol **163** (1983), no. 2, 171–6.

[HR87] C. B. Harley and R. P. Reynolds, *Analysis of e. coli promoter sequences.*, Nucleic Acids Res **15** (1987), no. 5, 2343–2361.

[Pri75] D. Pribnow, *Nucleotide sequence of an rna polymerase binding site at an early t7 promoter.*, Proc Natl Acad Sci U S A **72** (1975), no. 3, 784–788.

[Ren61] A. Renyi, *On measures of entropy and information*, Proceedings of the 4th Berkeley Symposium on Mathematical Statistics and Probability, vol. 1, 1961, pp. 547–561.

[Rod06] E. A. Rodland, *Exact distribution of word counts in shuffled sequences*, Adv. in Appl. Probab. **38** (2006), no. 1, 116–33.